THE WALKING BUG

Developer Manual

The Walking Bug - 2019-03-21

Document information

Version	0.16.0
Drafting	Enrico Sanguin Nicola Abbagnato
Check	Elisa Cattaneo Francesco De Filippis
Manager	Marco Dugatto
Use	External
Distribution	Prof. Tullio Vardanega Prof. Riccardo Cardin The Walking Bug
State	Approved

Description

This document defines a guide meant to be read by a developer who wants to maintain or extend Soldino.

Diary of changes

Version	Change	Author	Role	Date
1.0.0	*Approval of document for the RA*	Marco Dugatto	Manager	2019-05-09
1.0.0	*Positive verification of the document*	Elisa Cattaneo	Verifier	2019-05-09
0.20.0	*Verified §5.2.2, §5.2.3 §5.2.5 §5.2.7 §5.2.8*	Francesco De Filippis	Verifier	2019-05-08
0.20.0	*Updated §5.2.2, §5.2.3 §5.2.5 §5.2.7 §5.2.8*	Nicola Abbagnato	Developer	2019-05-08
0.19.0	*Verified §5.3*	Elisa Cattaneo	Verifier	2019-05-07
0.19.0	*Added the latest methods of SoldinoAPI*	Nicola Abbagnato	Developer	2019-05-05
0.18.0	Modified stucture of §5.3	Nicola Abbagnato	Developer	2019-05-01
0.17.0	Verified document	Francesco De Filippis	Verifier	2019-04-10
0.17.0	*Made defect corrections reported in RQ related to §1-§3*	Enrico Sanguin	Developer	2019-04-26
0.17.0	*Made defect corrections reported in RQ related to §4-§6*	Nicola Abbagnato	Developer	2019-04-25
0.16.0	*Approval of document for the RP*	Marco Dugatto	Manager	2019-04-15
0.15.6	*Positive verification of the document*	Francesco De Filippis	Verifier	2019-04-15
0.15.6	*Fixed errors in glossary*	Enrico Sanguin	Developer	2019-04-14
0.15.4	*Verified glossary*	Elisa Cattaneo	Verifier	2019-04-14
0.15.4	*Fixed errors in §5 and §6*	Enrico Sanguin	Developer	2019-04-14
0.15.0	*Verified §5 and §6*	Elisa Cattaneo	Verifier	2019-04-14
0.15.3	*Fixed errors in §4*	Enrico Sanguin	Developer	2019-04-13
0.15.0	*Verified §4*	Francesco De Filippis	Verifier	2019-04-12
0.15.2	*Fixed errors in §3*	Enrico Sanguin	Developer	2019-04-11
0.15.1	*Fixed errors in §1 and §2*	Enrico Sanguin	Developer	2019-04-10
0.15.0	*Verified §1, §2, §3*	Francesco De Filippis	Verifier	2019-04-10
0.15.0	*Updated glossary*	Enrico Sanguin	Developer	2019-04-09

0.14.0	*Written §4.6, §4.7*	Enrico Sanguin	Developer	2019-04-09
0.13.0	*Updated glossary*	Nicola Abbagnato	Developer	2019-04-09
0.12.0	*Written §5.3*	Nicola Abbagnato	Developer	2019-04-08
0.11.0	*Written §5.4*	Nicola Abbagnato	Developer	2019-04-08
0.10.0	*Updated glossary*	Enrico Sanguin	Developer	2019-04-08
0.9.0	*Written §4.2, §4.3, §4.4, §4.5*	Enrico Sanguin	Developer	2019-04-07
0.8.0	*Written §4.1*	Enrico Sanguin	Developer	2019-04-07
0.7.0	*Updated glossary*	Enrico Sanguin	Developer	2019-04-06
0.6.0	*Written §5.2.6, §5.2.7, §5.2.8 and §5.2.9*	Nicola Abbagnato	Developer	2019-04-21
0.5.0	*Written §5.1, §5.2.1, §5.2.2, §5.2.3, §5.2.4 and §5.2.5*	Nicola Abbagnato	Developer	2019-04-06
0.4.0	*Written §5.1*	Nicola Abbagnato	Developer	2019-04-06
0.3.0	*Written §2, §3 and §6*	Nicola Abbagnato	Developer	2019-04-05
0.2.0	*Writing §1*	Enrico Sanguin	Developer	2019-04-04
0.1.0	*Set up document layout*	Enrico Sanguin	Developer	2019-04-04

Contents

List of Figures

List of Tables

1 Introduction

1.1 Document purpose

The Developer Manual helps the developer to get access to all the functionality and methods of Soldino, assuring a complete knowledge of every technology used in this project in order to dispel any kind of doubt.

1.2 Product purpose

The purpose of the product is the creation of a DAPP on the Ethereum network and available from Mozilla Firefox 9.1 and Google Chrome 71 usable with the plugin MetaMask and accessible through a UI.
Soldino expects three user types and their connected features:

- **Goverment**:
  - Mint and distribute Cubit;
  - Require tax payment and check status;
  - Manage the business list.
- **Business Owner**:
  - Register their Business to the Government list;
  - Manage their goods and services;
  - Trade with other businesses;
  - Manage taxes.
- **Citizen**:
  - Trade with businesses.

2 Getting Started

2.1 Requirements

2.1.1 User requirements

To be able to use the product once it's running on a public blockchain like ropsten, rinkeby or the mainnet it's sufficient to have either Google Chrome 71+ or Mozilla Firefox 9.1+ with the Metamask extension installed.

- **Windows**:
  - **CPU**: Pentium 4 or newer with SSE2;
  - **Memory (RAM)**: 512 MB / 2G of RAM for the 64-bit version;
  - **Hard disk drive free space**: 200 MB;
  - **Operating System version**: Windows 7, 8, 10, or newer.
- **OS X**:
  - **CPU**: Any Intel CPU;
  - **Memory (RAM)**: 512 MB;
  - **Hard disk drive free space**: 200 MB;
  - **Operating System version**: OS X 10.9 or newer.
- **Linux**:
  - **CPU**: Pentium 4 or newer with SSE2;
  - **Memory (RAM)**: 512 MB;
  - **Hard disk drive free space**: 200 MB;
  - **Operating System version**: Kernel 2.2.14 or newer;
  - **Required libraries or packages (or newer)**: GTK+ 3.4, GLib 2.22, Pango 1.14, X.Org 1.0 (1.7 or higher is recommended), libstdc++ 4.6.1;
  - **Recommend libraries or packages**: NetworkManager 0.7 or higher, DBus 1.0 or higher, GNOME 2.16 or higher, PulseAudio.

2.1.2 Developer requirements

To be able to modify and test the product it's also necessary to install Npm, Git and possess at least 1GB of free hard disk drive space. To check if the installation of Npm and Git is successfull run the following commands:

```
node -v
npm -v
git -version
```

2.2 Installation

1. **If on windows**: run this command to install Python and some utilities that are necessary to build the application
   `npm install --global --production windows-build-tools`
2. Clone or download the repository at this link:
   https://github.com/frncscdf/The-Walking-Bug.git;
3. run this command from inside the repository folder:
   `npm install`

2.3 Setting up working enviroment

The team used WebStorm[1] to develop the application, with the following plugins:

- BashSupport: to create and edit the .sh scripts used to manage the contracts
- Intellij-Solidity: to create and edit the smart contracts
- Solidity Solhint: to execute the linting of the smart contracts at the moment of writing

The configuration options used are the following (REPOSITORY_FOLDER is the folder where the github repository was cloned):

- Languages & Frameworks > JavaScript > Code Quality Tools > ESLint:
  - Manual ESLint configuration;
  - Node interpreter: Project;
  - ESLint package: REPOSITORY_FOLDER/node_modules/eslint;
  - Configuration File: REPOSITORY_FOLDER/.eslintrc.json.
- Languages & Frameworks > JavaScript > Libraries: The following libraries were downloaded and enabled:
  - Chai;
  - Chai-as-promised.
- Tools > Solhint:
  - Node Binary: folder in which npm was installed
  - Solhint JS File: REPOSITORY_FOLDER/node_modules/solhint/solhint.js

2.4 How-to

2.4.1 Run on a local blockchain

To run the Dapp on a local blockchain the user needs to:

[1] https://www.jetbrains.com/webstorm/

1. Start the local blockchain with ganache-cli (already installed) with the following command:
   `ganache-cli`[2]
   or with Ganache[3];
2. Add the network to the truffle-config.js file;
3. Build the contracts with the following command:
   `npm run buildB -- NETWORK_NAME GOVERNMENT_ADDRESS]`
   - NETWORK_NAME is the name of the network found in truffle-config.js;
   - GOVERNMENT_ADDRESS is the address of the user in the local network who will be the government user;

2.4.2 Run the webApp on localhost

To run the webApp on localhost it's sufficient to run the following command:
`npm run start`

2.4.3 Test the backend

To test the smart contracts it's sufficient to run the following command:
`npm run testB`
To calculate the code coverage of the smart contracts it's sufficient to run the following command:
`npm run coverageB`
this will also automatically run the tests.

2.4.4 Update the contracts already running on a blockchain

To update the logic of the contracts already running on a blockchain it's sufficient to run the following command:
`npm run update -- NETWORK_NAME GOVERNMENT_ADDRESS]`

- NETWORK_NAME is the name of the network found in truffle-config.js;
- GOVERNMENT_ADDRESS is the address of the government user;

Only the contracts whose code was changed will be updated.

[2] https://github.com/trufflesuite/ganache-cli#using-ganache-cli
[3] https://truffleframework.com/ganache

3 Architecture

3.1 Introduction

In this section will be described the architecture overview of Soldino in order to have an high level view of the whole system.

3.2 Architecture overview

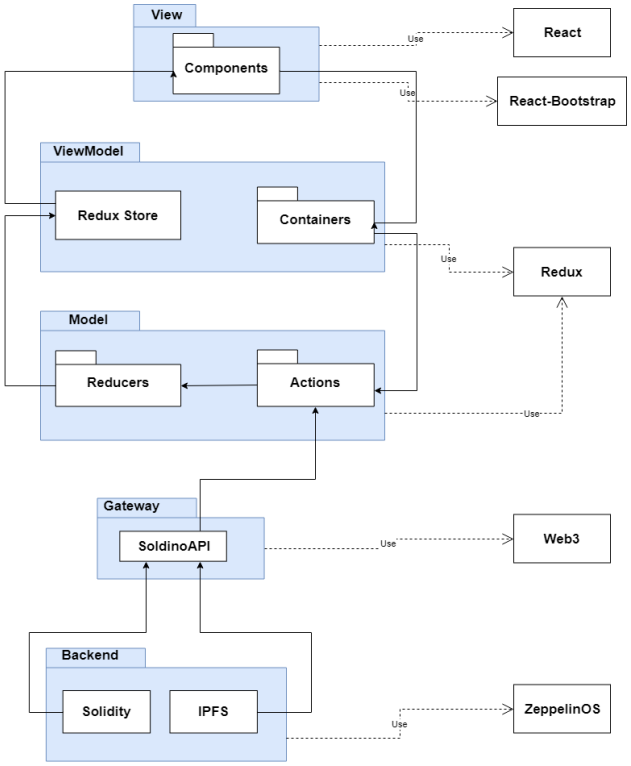

Figura 1: Architecture overview

3.3 Architecture description

The architecture in the picture shown above consist of three main parts. At the very bottom there is the backend part, that is a *Monolithic architecture*$_{\text{g}}$, made of the contracts written in *Solidity*$_{\text{g}}$ and the *IPFS*$_{\text{g}}$ used for the storage part of the data outside the blockchain.
The backend communicate with outside through a gateway named SoldinoAPI, a library written in *Web3*$_{\text{g}}$ that expose all the methods that an external application can use in order to interact with the blockchain and IPFS.
The last part of the architecture is the frontend one, that takes the role of a *microservice*$_{\text{g}}$, that use the *MVVM*$_{\text{g}}$ design pattern. This part is the representation of Soldino *WebApp*$_{\text{g}}$.
In this diagram are also illustrated the most important dependencies that the whole architecture use. The details of each part of the architecture mentioned above will be covered in the next sections of this document.

4 Frontend

4.1 Introduction

The purpose of this section is to take an in depth look to the MVVM pattern used for design the WebApp structure.

4.2 View

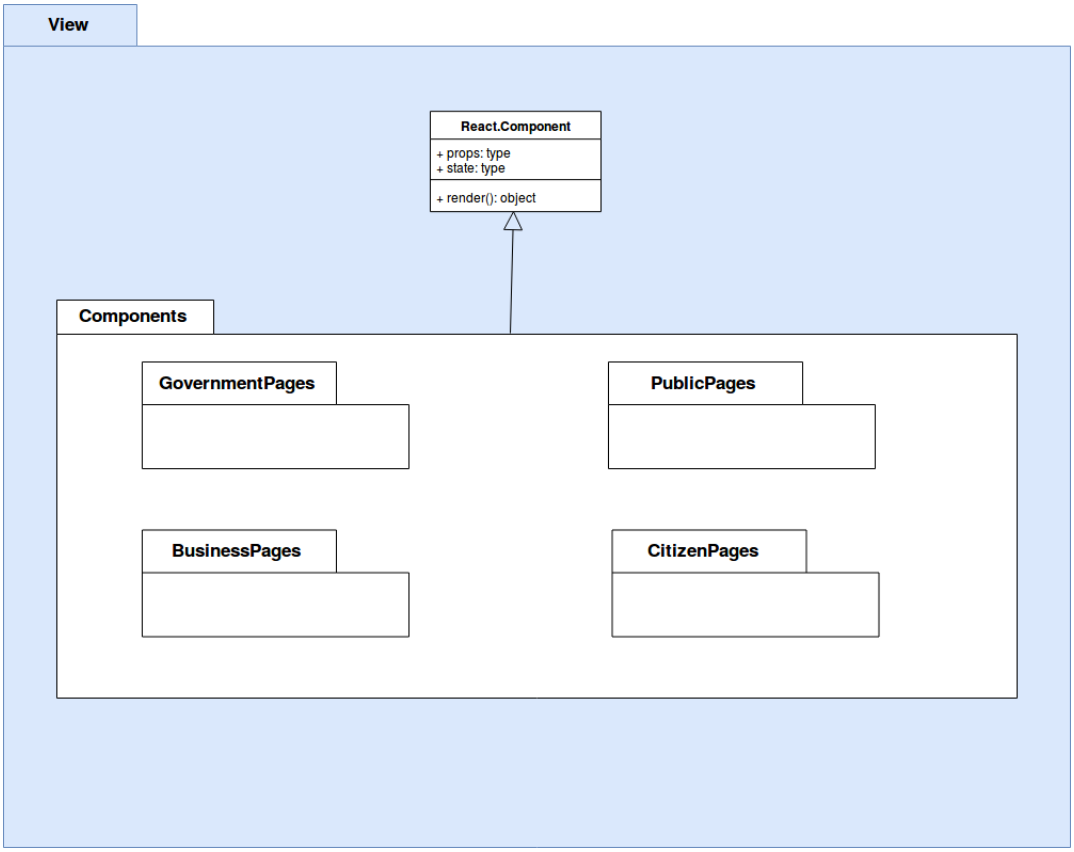

Figura 2: View layer

The $View_g$ layer showed in the picture above, concern of all the parts of the system whose responsibility is to display the UI_g with the information for the user. This parts are made by $React_g$[4] $components_g$ which extends the class $Component_g$ available in the React package. The components are divided into four packages:

- **PublicPages**: contains all the components and pages in common between all types of users;
- **GovernmentPages**: contains all the components and pages for the government;
- **BusinessPages**: contains all the components and pages for the business owners;
- **CitizenPages**: contains all the components and pages for the citizens;

[4] https://reactjs.org/

4.2.1 Bootstrap components

All the Soldino components are made with BoostrapReact, an adaptation of the Bootstrap *toolkit*$_g$ used for develop UI elements for the web as quick as possible. In the diagram below are illustrated all the compoenents offered by bootstrap that we used in out custom components which will be shown in the next section.

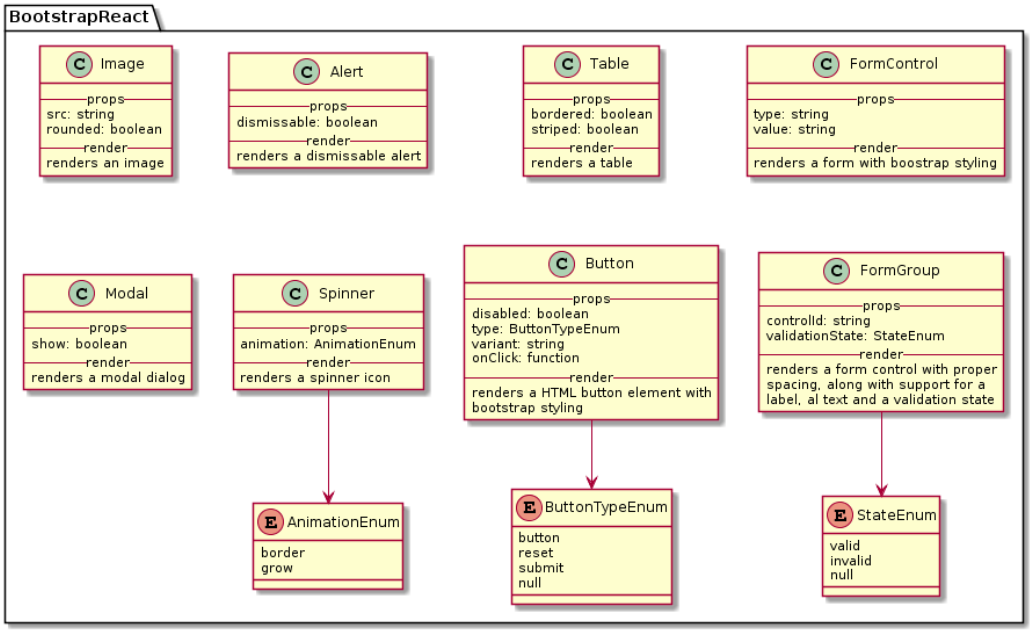

Figura 3: Bootstrap components

4.2.2 Custom components

In the diagram below are illustrated all the custom component made by us that use the BoostrapReact component that are shown in the diagram above.

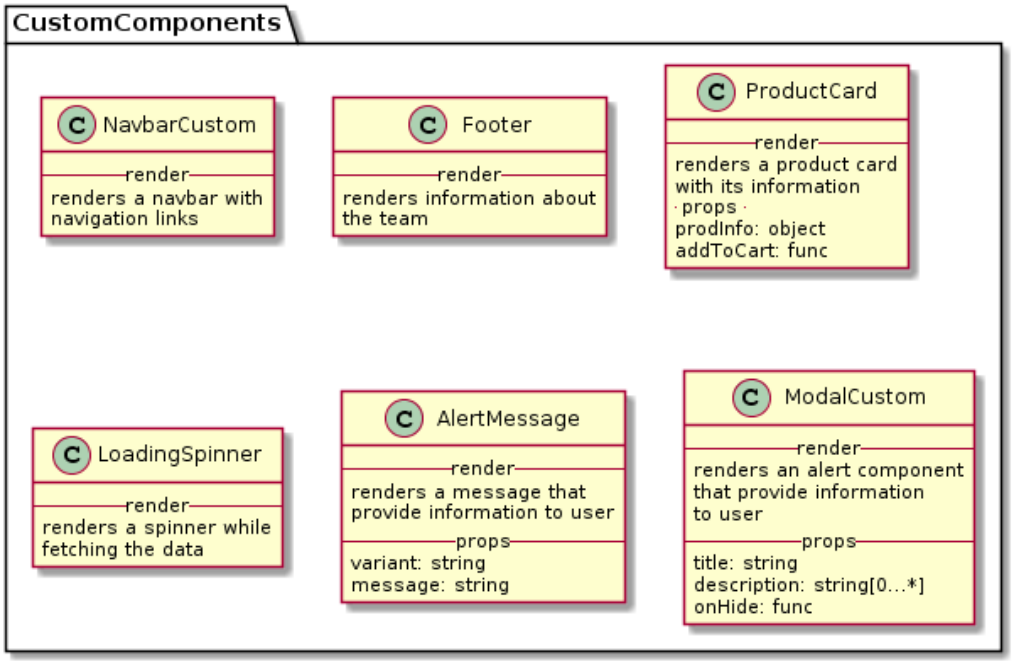

Figura 4: Custom components

4.2.3 Root component

The diagram below shows the Root component of Soldino and the the common components that it renders in all of the application pages. The Root component has the $Redux_g$ $store_g$ object and the $routes_g$ of all the applications. In the App component will be rendered all the children components according to the type of users and the route of the application. This component always has the footer, the navbar and the breadcrumbs.

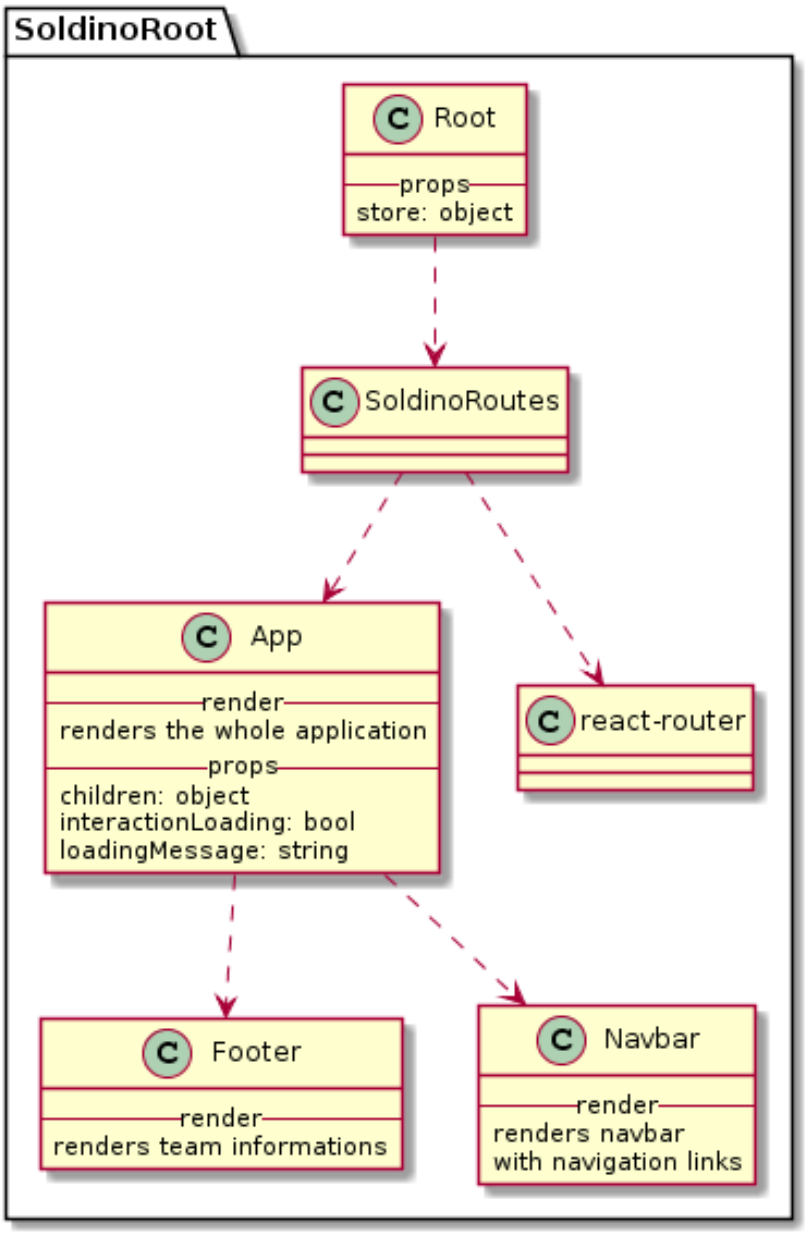

Figura 5: Root component

4.2.4 Public pages

In the diagram below are illustrated the public pages of Soldino.

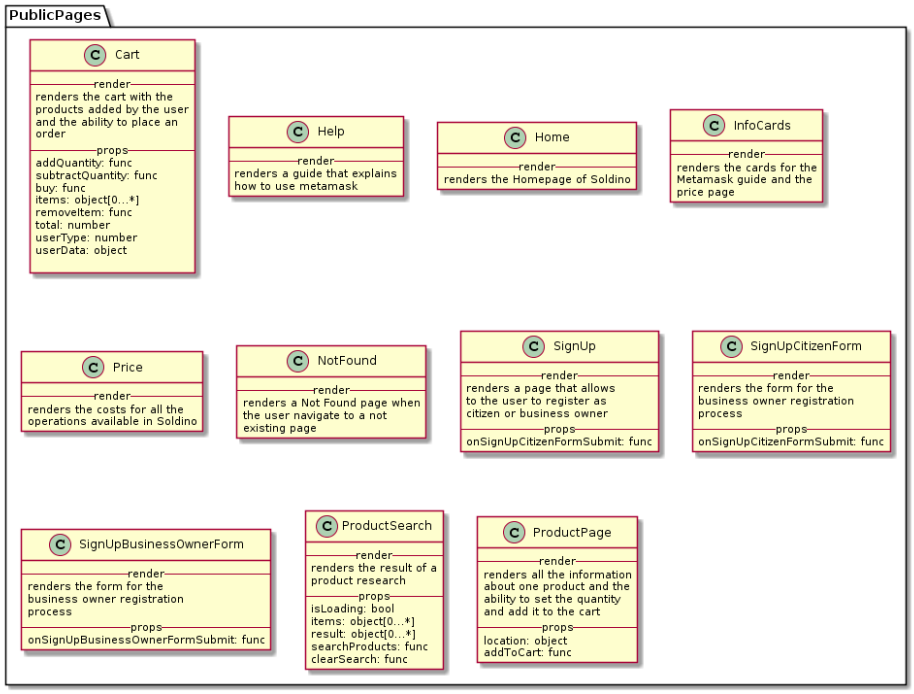

Figura 6: Public pages

4.2.5 Citizen pages

In the diagram below are illustrated the public pages of Soldino.

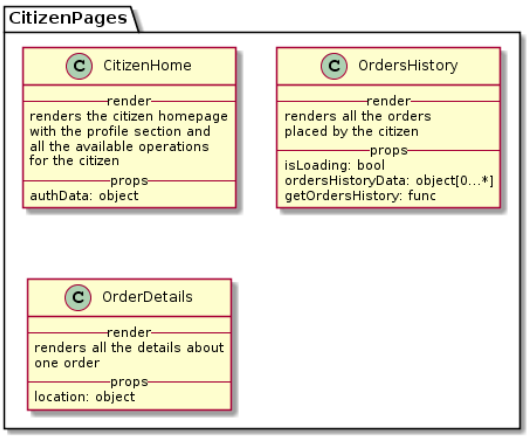

Figura 7: Citizen pages

4.2.6 Government pages

In the diagram below are illustrated the government pages of Soldino.

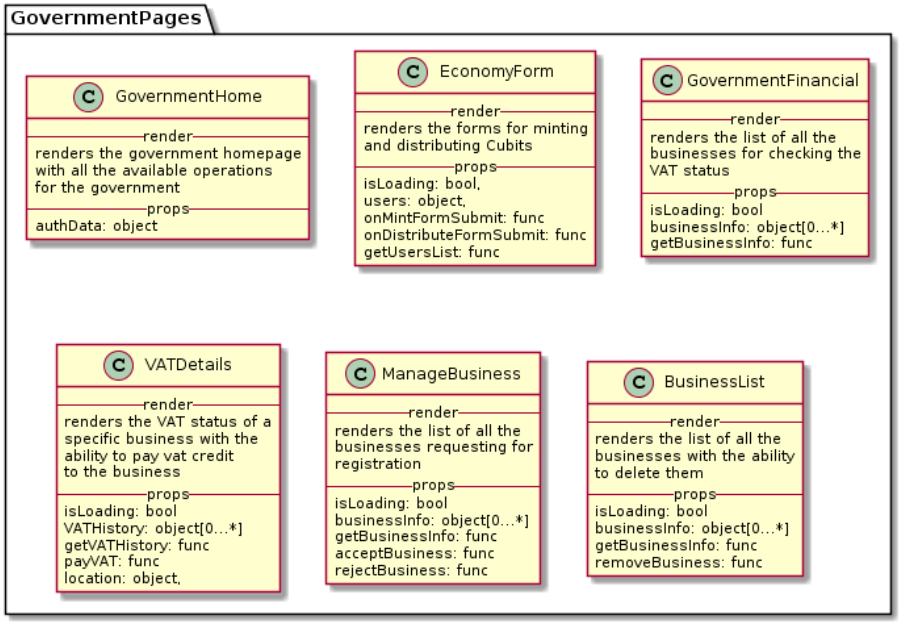

Figura 8: Government pages

4.2.7 Business owner pages

In the diagram below are illustrated the business owner pages of Soldino.

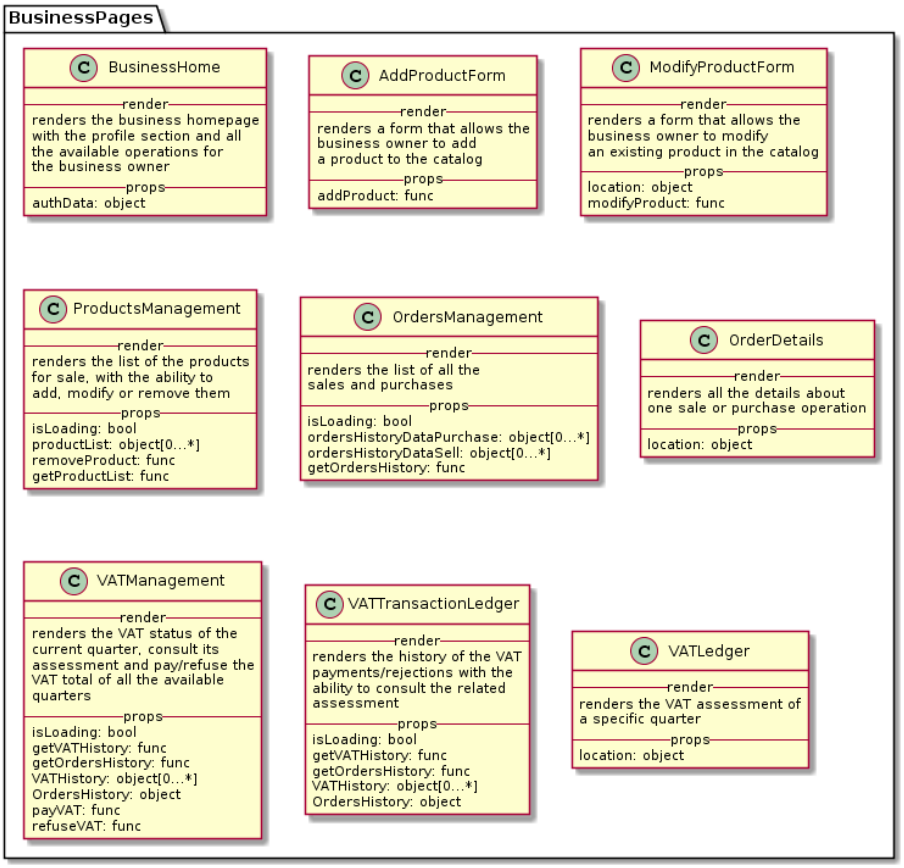

Figura 9: Business pages

4.3 ViewModel

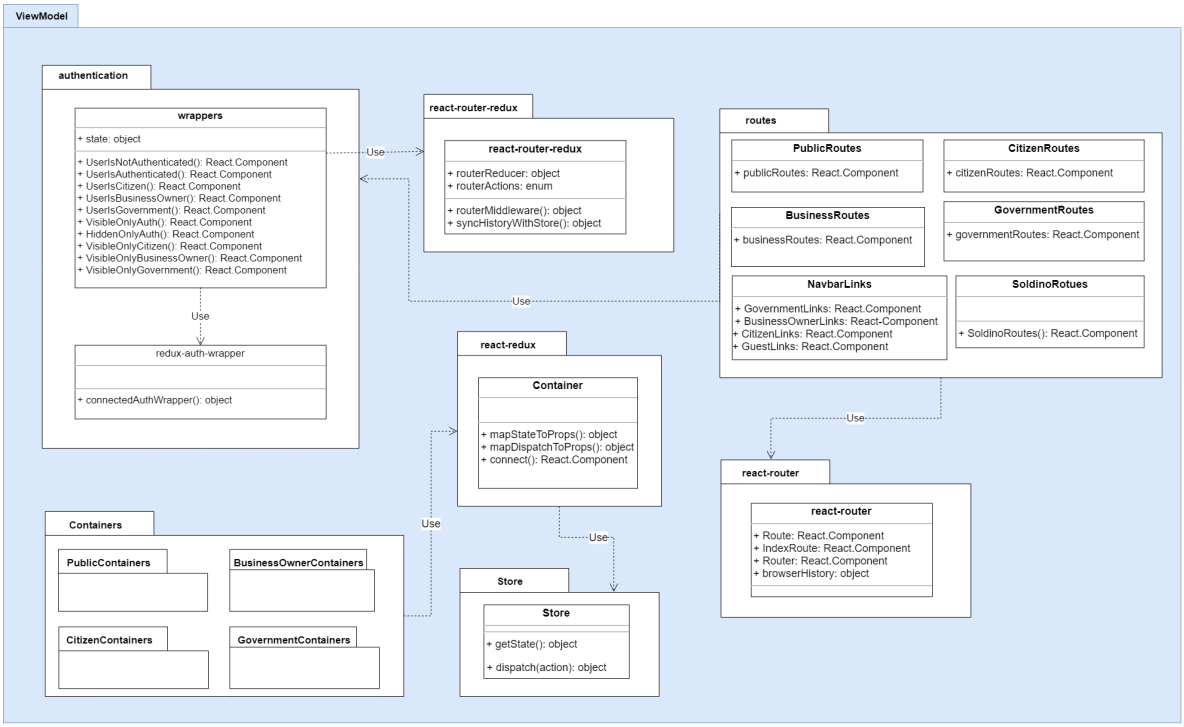

Figura 10: ViewModel layer

The *ViewModel*$_g$ layer showed in the picture above is responsible for bind all the view components with the data and functions placed in the *Model*$_g$ layer that will be described in the next section. In order to do this operation we used a common React-Redux pattern named Container-Component. This pattern consists of having a container associated with each component that needs to communicate with the data in the *store*$_g$ and use the *actions*$_g$ associated with the *reducers*$_g$. The Containers package is organised in the same way as the Components package in order to keep the folders structure clean and clear. As can be seen from the diagram, each container has always the same three methods, for this reason we will not show all the diagrams for every single containers because they would be indentical. In order to decide when and which component the system should render according to the user type, there is a wrapper class that offers the methods needed. This class use redux-auth-wrapper [5], an external library that allows to decouple the authentication business from the components.

[5] https://mjrussell.github.io/redux-auth-wrapper/

4.3.1 Routes

The diagram below shows how the routing in Soldino is made. The component SoldinoRoutes collects all the routes divided by the user type. In order to build the routes we used react-router [6] package that offers some useful components that allows the pages redirection. NavbarLinks class offers some methods that returns React components which are used by the NavbarCustom component in order to render the correct navigations links accoding to the user type.

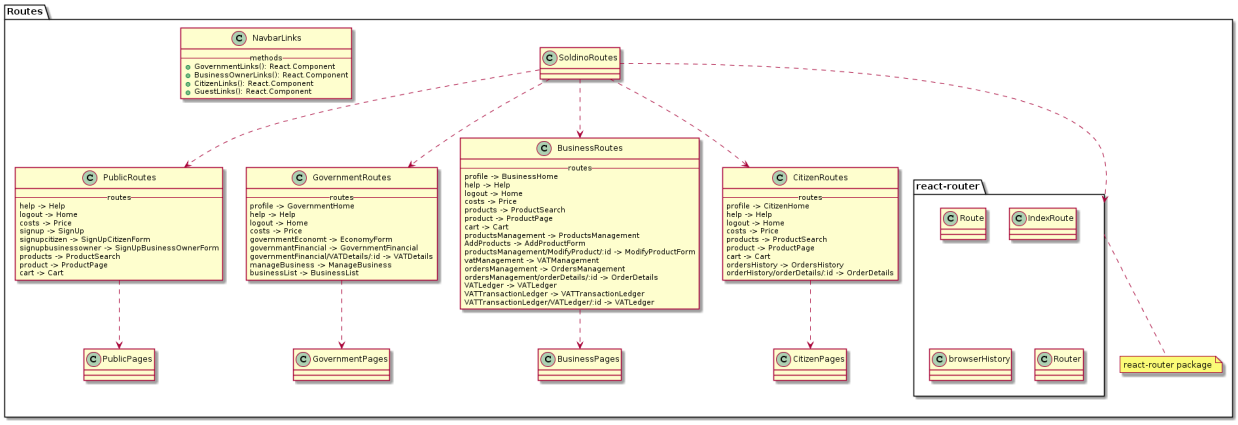

Figura 11: Routes

[6] https://reacttraining.com/react-router/

4.4 Model

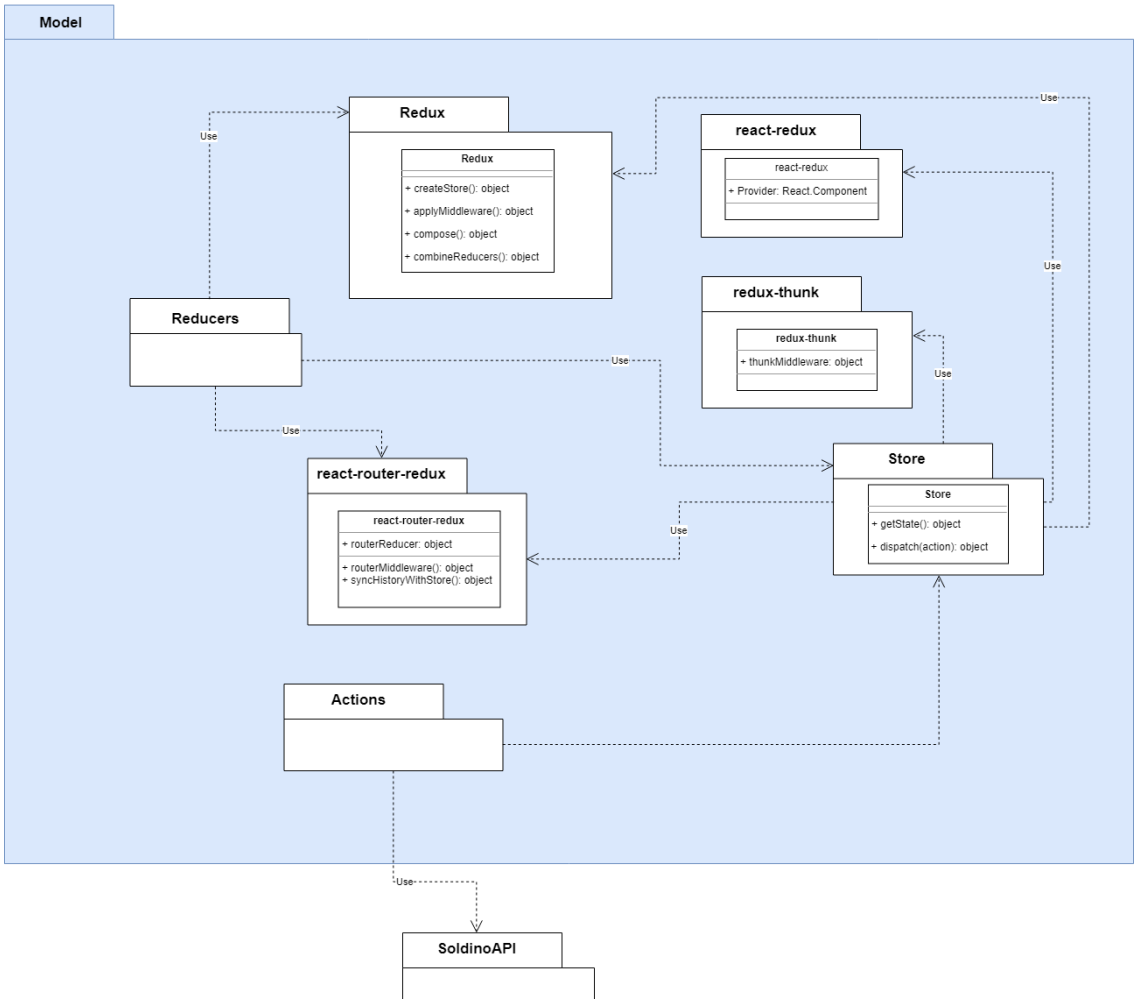

Figura 12: Model layer

The model layer showed in the picture above is responsible for manage all the data of the whole application and offers all the methods that are used for manipulate them. All the data is stored in the store object which is connected with the reducers and actions through some methods available in the *Redux*$_g$ [7] package. In the action package there are all the methods that fetch or send data to the blockchain through SoldinoAPI. When one of the actions are dispatched there is an equivalent reducer that is responsible of checking the action type and update the data of the store according to the payload that comes with the dispatched action.

[7] https://redux.js.org/

4.4.1 Public actions

In the diagram below are illustrated all of the actions which responsibility is to manage all the operations in common between all types of users.

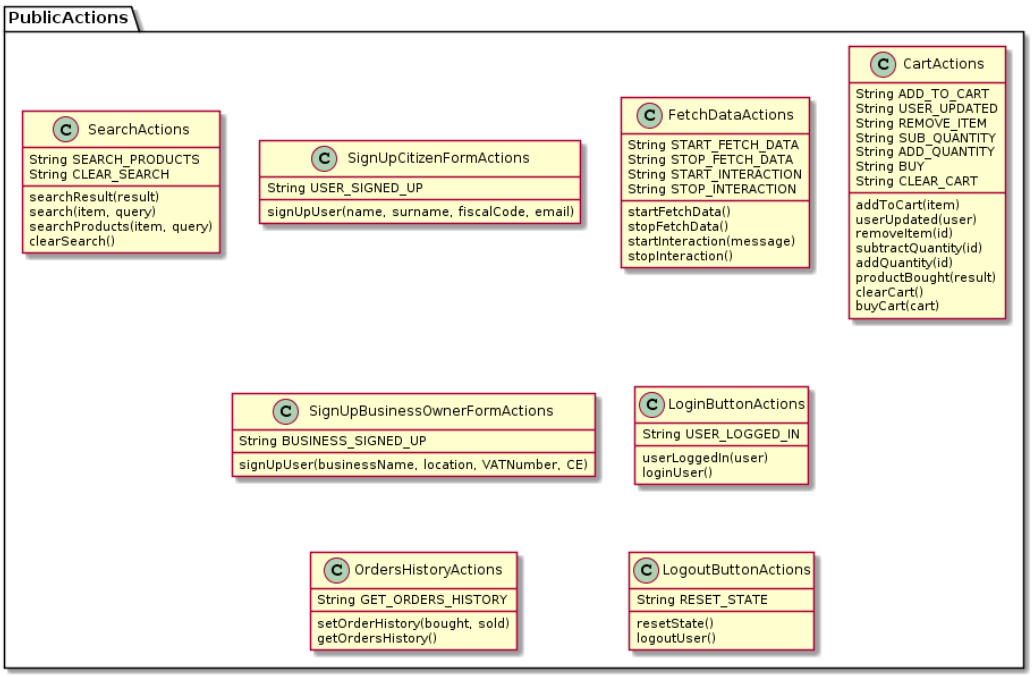

Figura 13: Common actions

4.4.2 Government actions

In the diagram below are illustrated all of the actions which responsibility is to manage all the government operations.

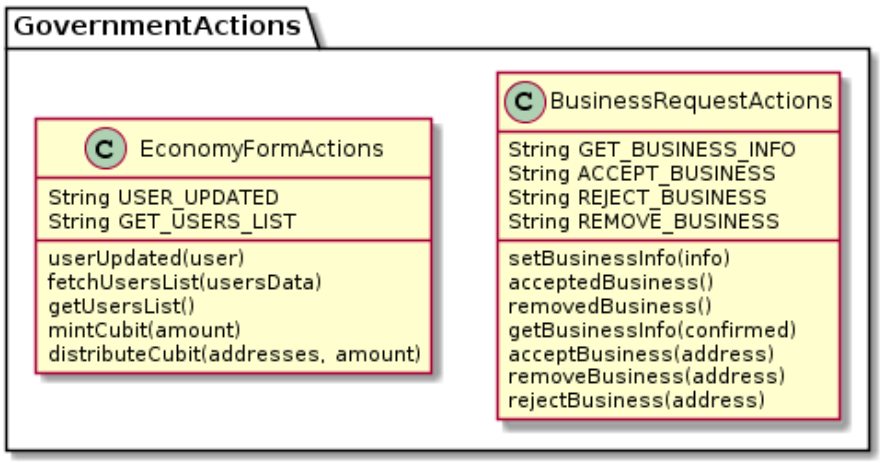

Figura 14: Government actions

4.4.3 Business Owner actions

In the diagram below are illustrated all of the actions which responsibility is to manage all the business owner operations.

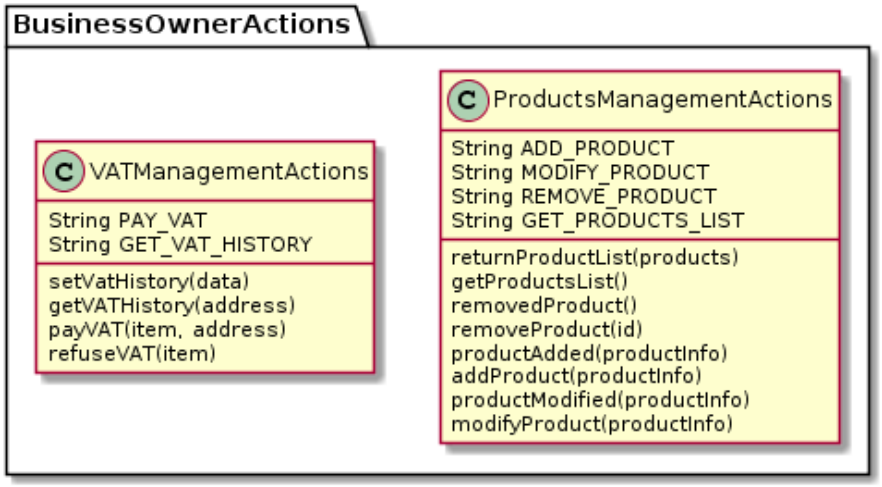

Figura 15: Business Owner actions

4.4.4 Citizen actions

All the citizen actions are included in the common actions package, for this reason no diagram is provided. If in the future it will be necessary to have specific actions for only citizens then it will be necessary to provide a specific package.

4.4.5 Reducers

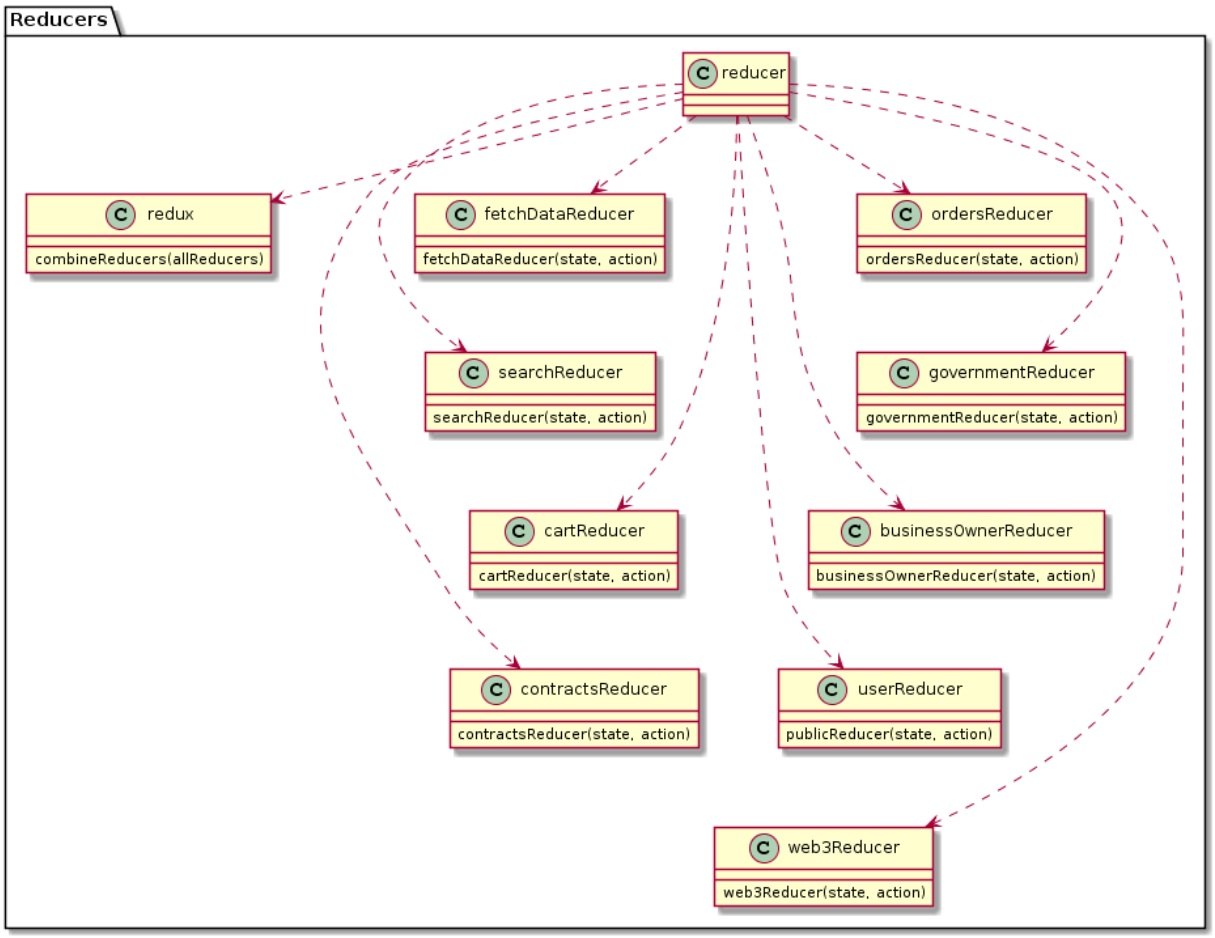

Figura 16: Reducers

4.5 Modify or add a feature to Soldino

4.5.1 Introduction

This section has the goal to provide a step-by-step guide in order to add new features or to modify the existing one of Soldino application. Since all the frontend of Soldino adopt the $MVVM_g$ pattern, this guide will be organised following the three layers already illustrated in the section above. Adding a new feature or modifying an existing one follows the same flow of operations, therefore we will describe the adding process.

4.5.2 View Layer

This layer concern of all the components that extends the React Component abstract class. The purpose of this components is to create web pages that renders static content or render a page with the data that is passed to this component as props. A react component should follow this instructions:

- The component should be located in the folder `src/components/<userType>` if it belongs to a specific user, otherwise in `src/components/public` if it is a component that is available to all type of users, even if not authenticated;
- Each component that receive data from props should have a `constructor(props)` that is used for recieve the props from the parent component and it is very useful to initialize some class fields. The constructor could also hold a state that is used for manage all the data that are not meant to be in the global store such as the fields information of a form.

In the picture below there is an example of a basic React component with its common properties:

```
1  import React from "react";
2
3  class NewComponent extends React.Component {
4    constructor(props) {
5      super(props);
6
7      this.propsFromParent = this.props;
8
9      this.state = {
10       compState: ""
11     }
12   }
13
14   someMethod() {}
15
16   render() {
17     //do stuff...
18     return (
19       <div>
20         This is a new React component with some data
21         from parent component
22         {this.propsFromParent.value} amd some data from
23         the state {this.state.compState}
24       </div>
25     );
26   }
27 }
```

Figura 17: New React component

4.5.3 ViewModel Layer

This layer has the responsibility to connect React components with *Redux*$_g$ *store*$_g$. In this way we can bind the data and actions defined in the Model with the components that will render the application data and will make available all the methods used for manipulate them.
In order to connect a React component with the store the instruction below should be followed:

- We need to create a new Container which will be situated in the folder `src/containers/<userType>` if it belongs to a specific user, otherwise in `src/container/public` if it is a container that is available to all type of users, even if not authenticated;
- Each container will have two methods: `mapStateToProps()` and `mapDispatchToProps()`. In order to connect the component previously defined with the store, we need to call the method `connect()`;
- Once our component has been connected we probably need to set up a Route for it. In the package routes there all the routes of Soldino, splitted in different file based on the user type. In the file relative to the Container-Component user we need to add a new Route component which is available from the react-router package. In order to manage the user permission we need to call the correct method defined in the package authentication (relative to the user type) and passing our Container. In this way our new container will be rendered if and only if this user type is authenticated.

In the pictures below there is an example of a basic all the steps listed above applied to a sign up form:

```
1  import { connect } from 'react-redux';
2  import SignUpCitizenForm from '../../components/public/SignUpCitizenForm';
3  import { signUpUser } from '../../actions/public/SignUpCitizenFormActions';
4
5  const mapStateToProps = (/* state, ownProps */) => ({});
6
7  const mapDispatchToProps = dispatch => ({
8    onSignUpCitizenFormSubmit: (name, surname, fiscalCode, email) => {
9      dispatch(signUpUser(name, surname, fiscalCode, email));
10   },
11 });
12
13 const SignUpCitizenFormContainer = connect(
14   mapStateToProps,
15   mapDispatchToProps,
16 )(SignUpCitizenForm);
17
18 export default SignUpCitizenFormContainer;
19
```

Figura 18: Container

```
import React from "react";
import { Route } from "react-router";
import { UserIsNotAuthenticated } from "../util/wrappers";

import Help from "../components/public/Help";
import SignUp from "../components/public/SignUp";
import Price from "../components/public/Price";
import SignUpCitizenFormContainer from "../containers/public/SignUpCitizenFormContainer";

export const publicRoutes = (
  <div>
    <Route path="signup" component={UserIsNotAuthenticated(SignUp)} />
    <Route path="costs" component={Price} />
    <Route path="help" component={Help} />
    <Route path="signupcitizen"
    component={UserIsNotAuthenticated(SignUpCitizenFormContainer)}/>
  </div>
);

```

Figura 19: Routes for sign up

4.5.4 Model Layer

This layer has the responsibility to store all the data of the application and it provide all the methods needed in order to manipulate them. In order to add a new feature the following instructions should be followed:

- First of all we need to add a new action which will be situated in `src/actions/<userType>` if it belongs to a specific user, otherwise in `src/actions/public` if it is an action that is available to all type of users, even if not authenticated;
- The action will perform a call to some methods of SoldinoAPI which communicate with the *Ethereum*$_g$ *Blockchain*$_g$ and *IPFS*$_g$. The action could fetch or send some data from them. The connection between this action and the *UI*$_g$ elements that triggers this methods is made in the *ViewModel*$_g$ layer as described in the previous section;
- Once we have created a new action, we need to add a new case to the corresponding reducer situated in `src/reducers/<userType>` if it belongs to a specific user, otherwise in `src/reducers/public` if it is a reducer that is available to all type of users, even if not authenticated. This allows the store to update the fields data when action is dispatched according to the payload send with the action itself.

In the pictures below there is an example of all the steps listed above applied to a login form:

```
import { browserHistory } from "react-router";
import { login } from "../../util/web3/SoldinoAPI";

export const USER_LOGGED_IN = "USER_LOGGED_IN";
function userLoggedIn(user) {
  return {
    type: USER_LOGGED_IN,
    payload: user
  };
}

export function loginUser() {
  return function(dispatch) {
    login().then(
      result => {
        console.log(result);
        dispatch(userLoggedIn(result));
        const currentLocation = browserHistory.getCurrentLocation();

        if ("redirect" in currentLocation.query) {
          return browserHistory.push(
            decodeURIComponent(currentLocation.query.redirect)
          );
        }
        return browserHistory.push("/profile");
      },
      err => {
        console.log(err);
      }
    );
  };
}

```

Figura 20: Action

```
1  const initialState = {
2    data: null,
3    type: "0"
4  };
5
6  const userReducer = (state = initialState, action) => {
7    if (action.type === "USER_LOGGED_IN" || action.type === "USER_UPDATED") {
8      return Object.assign({}, state, {
9        data: action.payload,
10       type: action.payload.type
11     });
12   }
13
14   if (action.type === "USER_LOGGED_OUT") {
15     return Object.assign({}, state, {
16       data: null,
17       type: "0"
18     });
19   }
20
21   return state;
22 };
23
24 export default userReducer;
25
```

Figura 21: Reducer

5 Backend

5.1 System Architecture

5.1.1 Overview

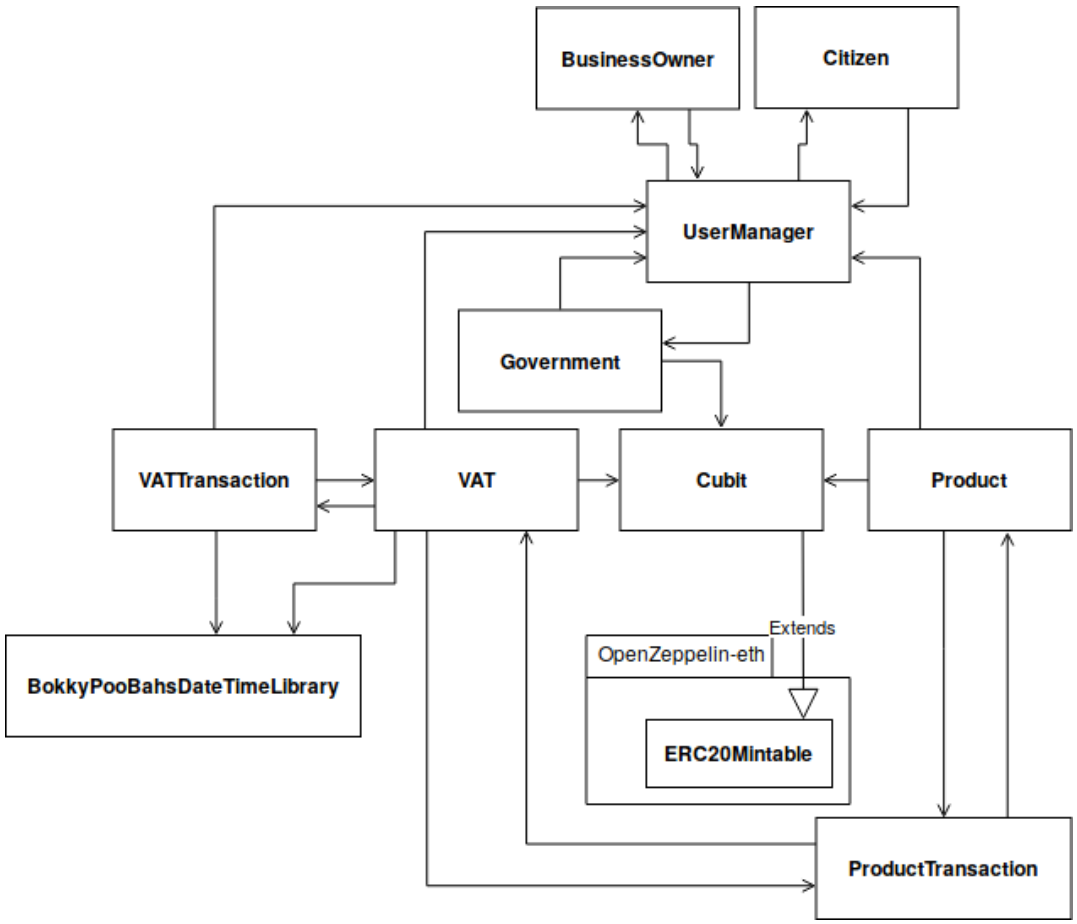

Figura 22: Backend architecture

The figure shown above illustrates the architecture of the Soldino application, composed of smart contracts which can be divided into three categories:

- Economic contracts;
- User management contracts;
- Transaction data storage contracts.

To achieve the upgradability of the contracts The-Walking-Bug used the "ZeppelinOS" framework; this automatically applies the destructured storage proxy pattern to each of the contracts shown above. In order to allow that, each of our contracts needs to derive from the Initializable contract provided by "ZeppelinOS" (not shown in the above image to avoid unnecessary complexity).

To be able to easily manage dates, fundamental for the storage of transactions, the team used BokkyPooBahsDateTimeLibrary[8] and added the functions to manage quarters.

To make sure the $ERC20_g$ token is implemented safely, the team used the contract ERC20Mintable from OpenZeppelin-eth[9] a collection of upgradable contracts useful for the developement of smart contracts.

[8] https://github.com/bokkypoobah/BokkyPooBahsDateTimeLibrary
[9] https://github.com/OpenZeppelin/openzeppelin-eth

5.1.2 User Management Contracts

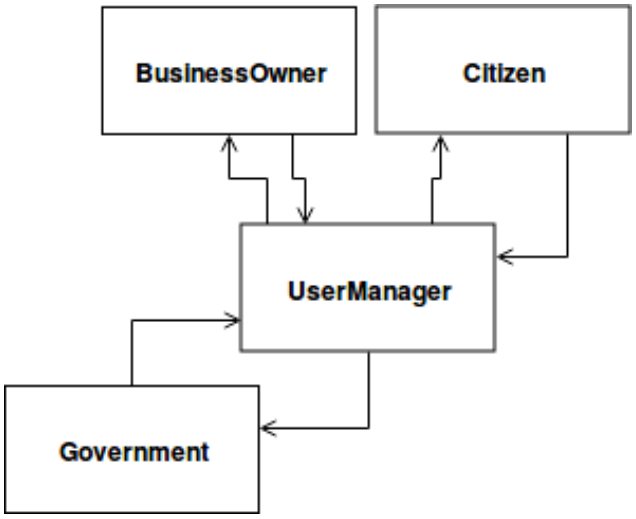

Figura 23: User management contracts architecture

As for the group of user management contracts we have a main contract, the UserManager, which handles calls to Citizen and BusinessOwner contracts entirely and also manages calls to Government authentication methods. This allows us to have a single management center for these calls and therefore reduce the coupling between the contracts of the economic part and those of the users.

5.1.3 Economic Contracts

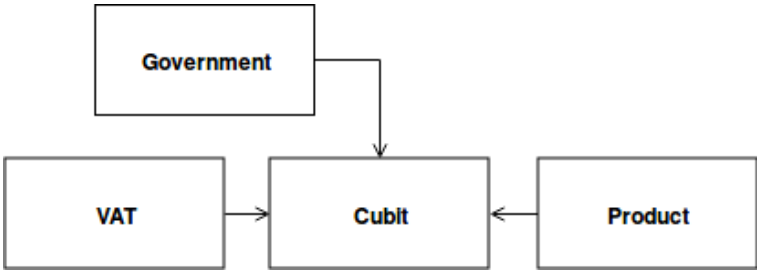

Figura 24: Economic contracts architecture

As regards the group of contracts of an economic nature, we have as a central contract "Cubit" our crypto-token ERC-20, which extends ERC20Mintable from OpenZeppelin-eth, which interacts with Government, VAT and Product contracts. Only these three contracts, in fact, have the ability to trigger a transaction and therefore a displacement of Cubit between two users. Cubit is a contract that derives from the "ERC20Mintable" library developed by "OpenZeppelin" which provides methods related to the crypto-token minting.

5.1.4 Transaction data storage contracts

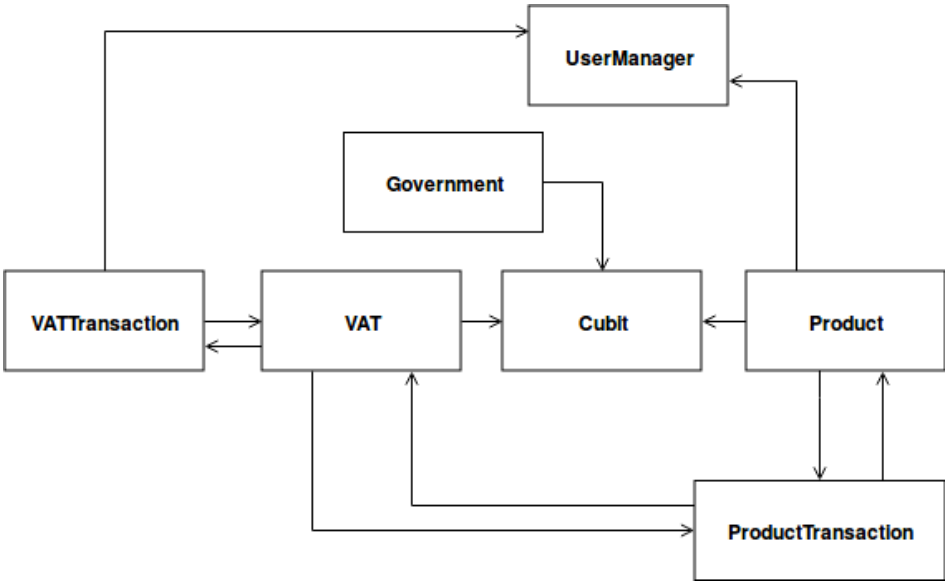

Figura 25: Transaction data storage contracts

The transaction information storage contracts, Product Transaction and VAT Transaction have the fundamental role of storing data related to e-commerce transactions and VAT adjustment respectively. By doing this we have the security of the inviolability of such data since they are stored in the blockchain.

5.2 Contracts

5.2.1 Cubit

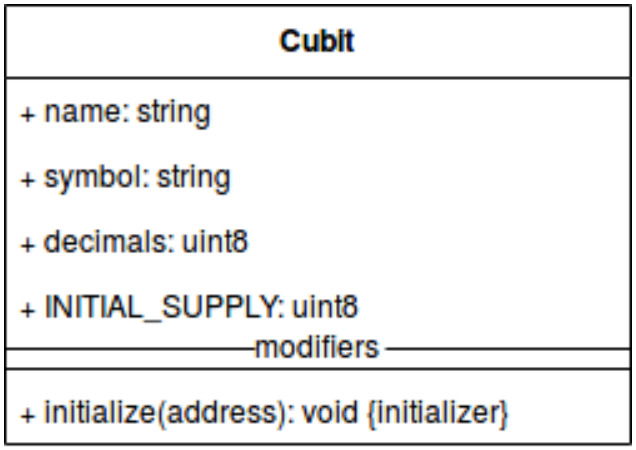

Figura 26: Cubit contracts

The Cubit contract is a simple implementation of the ERC20Mintable contract (from which it derives) developed by OpenZeppelin, which provides all the methods a crypto-token needs to adhere to the *ERC-20*$_g$ standard.

5.2.1.1 Attributes

Attribute name	Description
`name`	the name of the token
`symbol`	the symbol of the token
`decimals`	as ERC20 tokens are memorized as integers it represents the amount of decimal places after the comma
`INITIAL_SUPPLY`	the amount of tokens that will be minted as the contract is initialized

Tabella 2: Cubit contract attributes

5.2.1.2 Methods

Method signature	Description
`initialize(address minter)`	initializes the contract and mints INITIAL_SUPPLY tokens to the user at the given address

Tabella 3: Cubit contract methods

5.2.2 UserManager

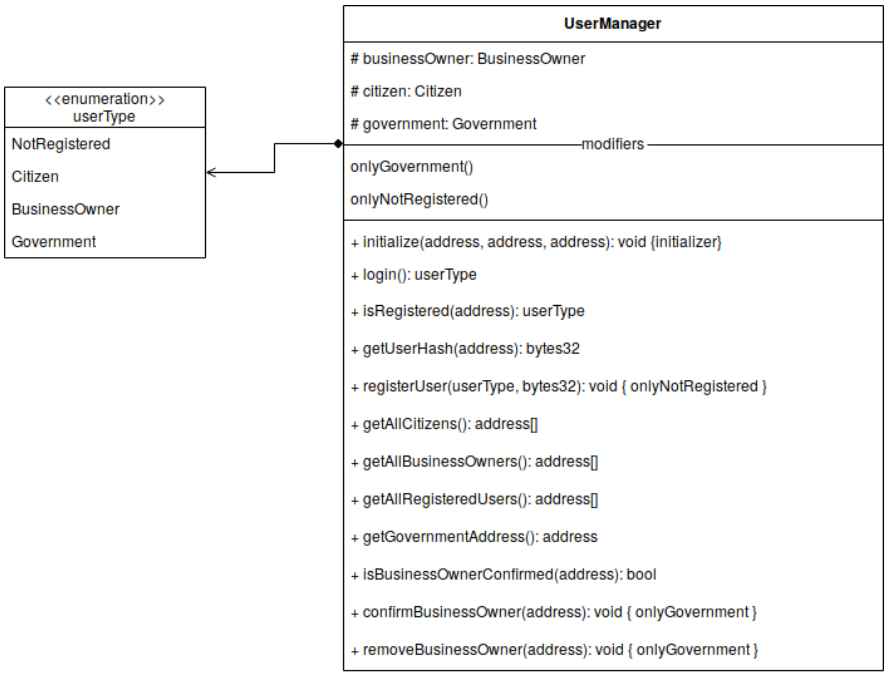

Figura 27: UserManager contract

The UserManager contract is the contract that manages calls to Citizen, BusinessOwner and Government contracts, the latter only for the authentication part, of which it has an instance. It provides login, registration, getter methods that provide user lists and others that verify the user status.

5.2.2.1 Attributes

Attribute name	Description
`businessOwner`	the instance of the deployed BusinessOwner's proxy
`citizen`	the instance of the deployed Citizen's proxy
`government`	the instance of the deployed Government's proxy

Tabella 4: UserManager contract attributes

5.2.2.2 Modifiers

Modifier signature	Description
`onlyGovernment()`	checks that only the government can access to a specific method
`onlyNotRegistered()`	checks that only not registered users can access to a specific method

Tabella 5: UserManager contract modifiers

5.2.2.3 Methods

Method signature	Description
`initialize(address _citizen, address _businessOwner, address _government)`	initializes the contract
`login()`	checks that the user who invoked the method is a registered user and returns the type of user
`isRegistered(address _user)`	checks that a specific address belongs to a registered user
`getUserHash(address _user)`	returns the IPFS hash belonging to a specific user
`registerUser(userType _type, bytes32 _hash)`	registers a user by calling the method based on `_type`

`getAllCitizens()`	calls the Citizen's method "getAll-Citizen"
`getAllBusinessOwners()`	calls the BusinessOwener's method "getAllBusinessOwners"
`getAllRegisteredUsers()`	returns the list of all registered users
`getGovernmentAddress()`	returns the address of the government user
`isBusinessOwnerConfirmed(address _businessOwner)`	checks if a specific business owner has been verified by the government
`confirmBusinessOwner(address _businessOwner)`	calls the BusinessOwener's method "confirmBusiness"
`removeBusinessOwner(address _businessOwner)`	calls the BusinessOwener's method "removeBusiness"

Tabella 6: UserManager contract methods

5.2.3 Government

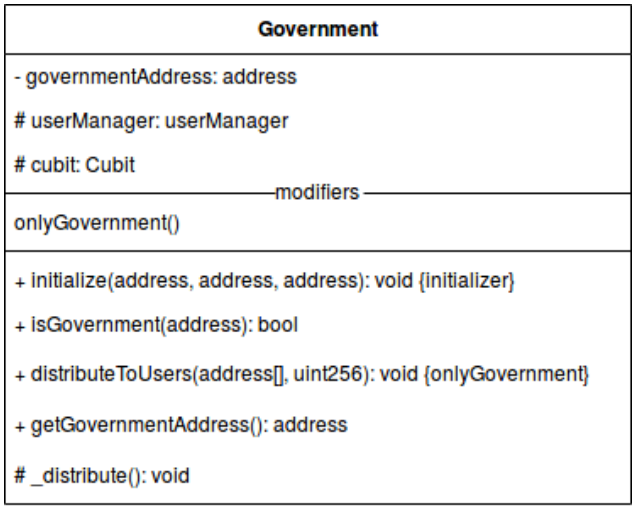

Figura 28: Government contract

The government contract manages what is the super user of our platform.

5.2.3.1 Attributes

Attribute name	Description
`governmentAddress`	the address of the government user
`userManager`	the instance of the deployed UserManager's proxy
`cubit`	the instance of the deployed Cubit's proxy

Tabella 7: Government contract attributes

5.2.3.2 Modifiers

Modifier signature	Description
`onlyGovernment()`	requires the calling user to be the government user

Tabella 8: Government contract modifiers

5.2.3.3 Methods

Method signature	Description
`initialize(address _userManager, address _cubit, address _governmentAddress)`	initializes the contract setting the address of the government to _governmentAddress
`isGovernment(address _user)`	returns true if the given user is the government user, otherwise returns false
`distributeToUsers(address[] memory _to, uint256 _amount)`	distributes the given amount of cubits to each user in the list, but only if that user is registered (and confirmed if the user is a business owner)
`getGovernmentAddress()`	returns the address of the government user
`_distribute(address[] _to, uint256 _amount)`	tries to distribute money to given a set of users

Tabella 9: Government contract methods

5.2.4 Citizen

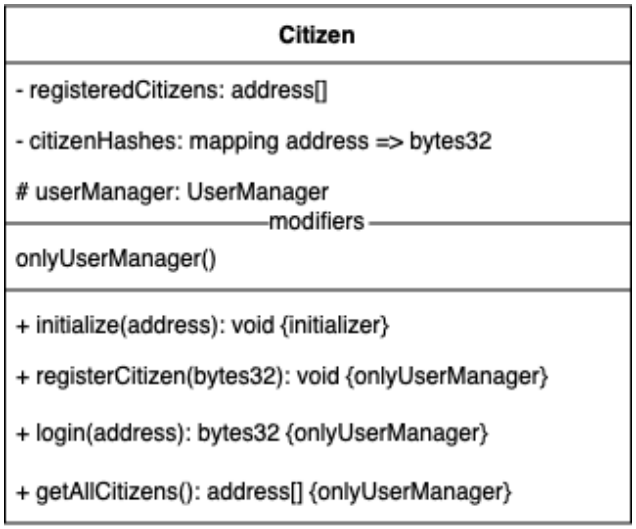

Figura 29: Citizen contract

The Citizen contract manages users who register as citizens in the platform, in fact it is equipped with a set of addresses, belonging to registered users, which allows to obtain the $JSON_g$ file containing the information of the aforementioned users from $IPFS_g$. Citizen has an instance of UserManager that allows it to check that its methods can only be called by the UserManager itself.

5.2.4.1 Attributes

Attribute name	Description
`registeredCitizens`	array containing the addresses of all citizens registered on the platform
`citizenHashes`	mapping containing the IPFS hashes associated of all citizens registered on the platform
`userManager`	the instance of the deployed UserManager's proxy

Tabella 10: Citizen contract attributes

5.2.4.2 Modifiers

Modifier signature	Description
`onlyUserManager()`	requires the UserManager contract to be the caller

Tabella 11: Citizen contract modifiers

5.2.4.3 Methods

Method signature	Description
`initialize(address _userManager)`	initializes the contract
`registerCitizen(bytes32 _hash, address _user)`	adds a citizen to the list of registered ones and map its address to the relative hash
`login(address _user)`	checks if the address belongs to a registered user
`getAllCitizens()`	returns the list of registered users

Tabella 12: Citizen contract methods

5.2.5 BusinessOwner

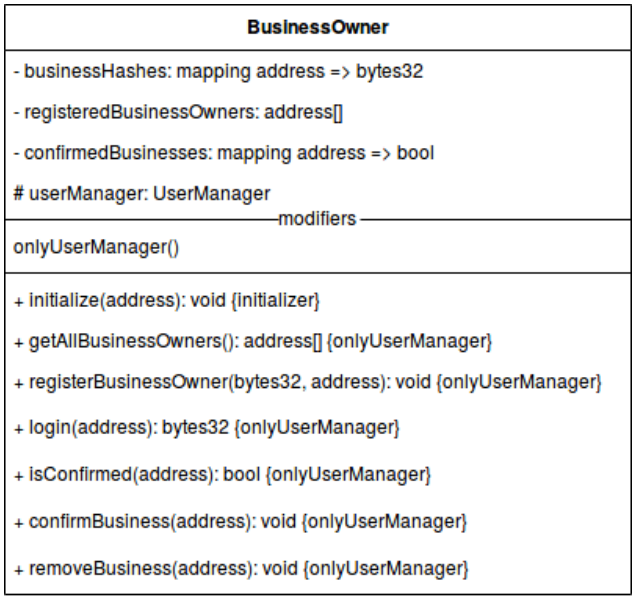

Figura 30: Citizen contract

The BusinessOwner contract manages users who register as an enterpreneur in the platform; in fact it is equipped with a set of addresses, belonging to registered enterpreneurs, which allows to obtain the $JSON_g$ file containing the information of the aforementioned users from $IPFS_g$. BusinessOwner has an instance of UserManager that allows it to check that its methods can only be called by the UserManager itself.

5.2.5.1 Attributes

Attribute name	Description
`governmentAddress`	the instance of the deployed Government's proxy
`userManager`	the instance of the deployed UserManager's proxy
`cubit`	the instance of the deployed Cubit's proxy

Tabella 13: BusinessOwner contract attributes

5.2.5.2 Modifiers

Modifier signature	Description
`onlyGovernment()`	checks that only the government can access to a specific method

Tabella 14: BusinessOwner contract modifiers

5.2.5.3 Methods

Method signature	Description
`initialize(address _userManager)`	initializes the contract
`registerBusinessOwner (bytes32 _hash, address _user)`	adds a business owner to the list of registered ones and map its address to the relative IPFS hash
`login(address _user)`	checks if the address belongs to a registered business owenr
`getAllBusinessOwners()`	returns the list of all business owners
`isConfirmed(address _businessOwner)`	checks if the address belongs to a confirmed business owner
`confirmBusiness(address _businessOwner)`	adds the address to the mapping of confirmed business owners
`removeBusiness(address _businessOwner)`	removes the address from the list of registered business owners and also from the confirmed ones mapping

Tabella 15: BusinessOwner contract methods

5.2.6 VAT

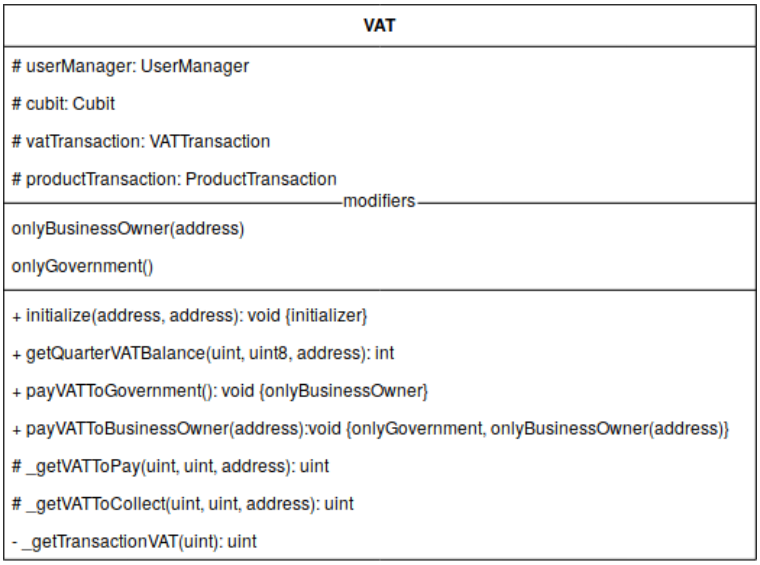

Figura 31: VAT contract

The VAT contract manages the VAT adjustment transactions carried out between the government and companies every quarter; it uses BokkyPooBahsDateTimeLibrary to manage the dates of the transactions.

5.2.6.1 Attributes

Attribute name	Description
`userManager`	the instance of the deployed UserManager's proxy
`cubit`	the instance of the deployed Cubit's proxy
`vatTransaction`	the instance of the deployed VatTransaction's proxy
`productTransaction`	the instance of the deployed ProductTransaction's proxy

Tabella 16: VAT contract attributes

5.2.6.2 Modifiers

Modifier signature	Description
`onlyBusinessOwner (address _businessOwner)`	requires the given address to be associated to a business owner
`onlyGovernment()`	requires the calling user to be the government user

Tabella 17: VAT contract modifiers

5.2.6.3 Methods

Method signature	Description
`initialize(address _cubit, address _userManager, _vatTransaction, _productTransaction, address _government)`	initializes the contract
`getQuarterVATBalance(uint256 _year, uint8 _quarter, address _businessOwner)`	returns the VAT balance of a given business owner in a given quarter
`payVATToGovernment()`	lets a business pay the VAT to the government
`payVATToBusinessOwner(address _businessOwner)`	lets the government refund the VAT to a given business owner
`_getVATToPay(uint256 _fromDate, uint256 _toDate, address _businessOwner)`	internal method that returns the negative VAT of a user from a given date to another
`_getVATToCollect(uint256 _fromDate, uint256 _toDate, address _businessOwner)`	internal method that returns the positive VAT of a user from a given date to another

Tabella 18: VAT contract methods

5.2.7 VATTransaction

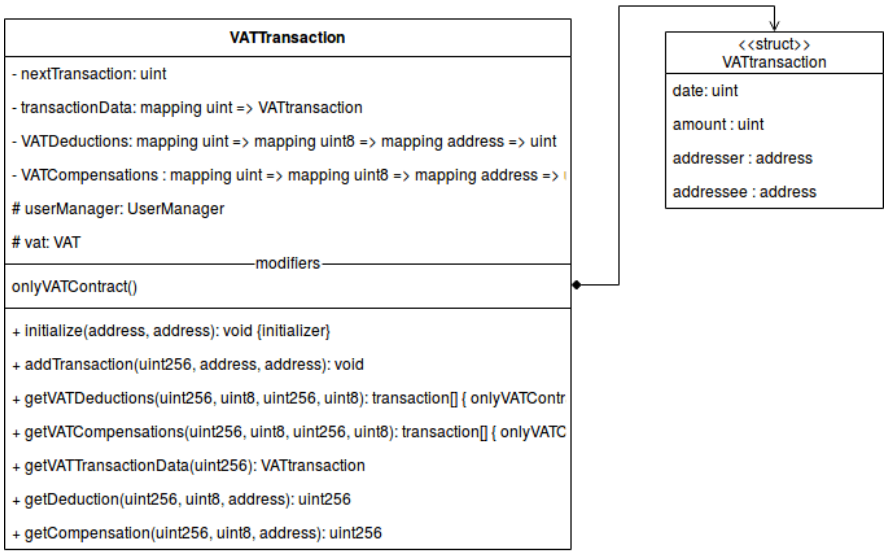

Figura 32: VATTransaction contract

VATTransaction is the contract in which the data related to VAT adjustment transactions, such as date, total cost, sender and recipient, are stored; it uses BokkyPooBahsDateTimeLibrary to manage the dates of the transactions.

5.2.7.1 Attributes

Attribute name	Description
`nextTransaction`	the ID of the next transaction to add
`transactionData`	a mapping that linking the ID of a transaction to it's data
`VATDeductions`	a mapping that stores if there was a VAT deduction from a certain business owner in a particular quarter of a certain year
`VATCompensations`	a mapping that stores if there was a VAT compensation to a certain business owner in a particular quarter of a certain year
`userManager`	the instance of the deployed UserManager's proxy
`vat`	the instance of the deployed VAT's proxy

Tabella 19: VATTransaction contract attributes

5.2.7.2 Modifiers

Modifier signature	Description
`onlyVATContract()`	requires the calling contract to be VAT

Tabella 20: VATTransaction contract modifiers

5.2.7.3 Methods

Method signature	Description
`initialize(address _userManager, address _vat)`	initializes the contract
`addTransaction(uint256 _amount, address _addresser, address _addressee)`	adds a transaction from a given business owner to the government or vice-versa
`getVATDeductions(uint256 _fromYear, uint8 _fromQuarter, uint256 _toYear, uint8 _toQuarter)`	returns all VAT deductions in a given time interval
`getVATCompensation(uint256 _fromYear, uint8 _fromQuarter, uint256 _toYear, uint8 _toQuarter)`	returns all VAT compensations in a given time interval
`getVATTransactionData(uint256 _transactionID)`	returns the data for the transaction linked to the given ID
`getDeduction(uint256 _fromYear, uint8 _fromQuarter, address _businessOwner)`	returns the ID of a deduction in the given quarter, for the given business
`getCompensation(uint256 _fromYear, uint8 _fromQuarter, address _businessOwner)`	returns the ID of a compensation in the given quarter, for the given business

Tabella 21: VATTransaction contract methods

5.2.8 Product

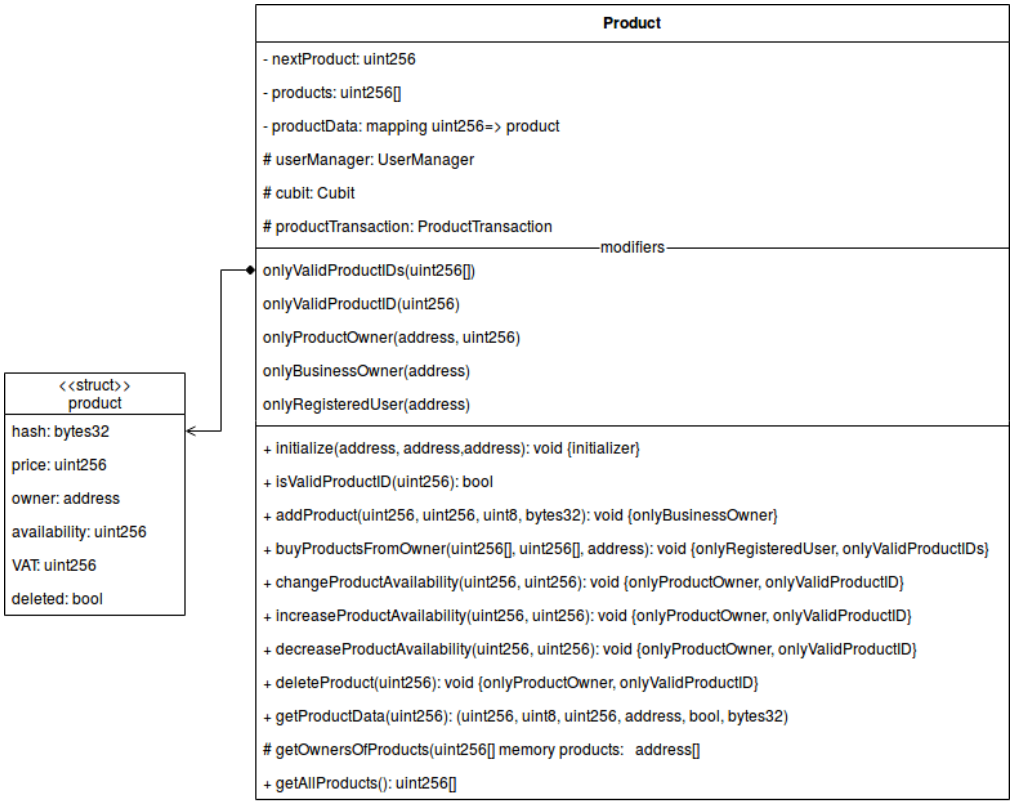

Figura 33: Product contract

5.2.8.1 Attributes

Attribute name	Description
`nextProduct`	counter that keeps track of the number of product. Is also used to understand the ID of the next product that will be inserted to the list of products
`products`	array that contains the IDs of all valid products in the platform
`productData`	mapping that stores data for all valid products in Soldino
`userManager`	the instance of the deployed UserManager's proxy
`cubit`	the instance of the deployed Cubit's proxy
`productTransaction`	the instance of the deployed ProductTransaction's proxy

Tabella 22: Product contract attributes

5.2.8.2 Modifiers

Modifier signature	Description
`onlyGovernment()`	checks that only the government can access to a specific method
`onlyValidProductIDs (uint256[] memory _products)`	checks that each ID in the array belongs to a valid product
`onlyValidProductID (uint256 _productID)`	checks that the ID belongs to a valid product
`onlyProductOwner (address _user, uint256 _productId)`	checks that only the product owner can access to a specific method
`onlyBusinessOwner (address _user)`	checks that the given address belongs to a business owner
`onlyRegisteredUser(address _user)`	checks that the given address belongs to a registered user

Tabella 23: Product contract modifiers

5.2.8.3 Methods

Method signature	Description
`initialize(address _userManager, address _cubit, address _productTransaction)`	initializes the contract
`isValidProductID(uint256 _productID)`	returns true if the given product ID belongs to a valid product ID
`addProduct(uint256 _price, uint256 _availability, uint8 _VAT, bytes32 _hash)`	adds a product to the product list

`buyProductsFromOwner(uint256[] memory _products, uint256[] memory _amount, address _owner)`	this method allow a registered user to buy a product from another registered user. Once the parameters have been checked for each product in _products, a transaction from the buyer to the seller will be triggered with the total amount of the goods sold
`getOwnersOfProducts(uint256[] _products)`	returns the owners of a set of products
`changeProductAvailability(uint256 _productId, uint256 _availabilityChange)`	allows to change a specific product's availability
`increaseProductAvailability(uint256 _productId, uint256 _availabilityChange)`	allows to increase a specific product's availability
`decreaseProductAvailability(uint256 _productId, uint256 _availabilityChange)`	allows to decrease a specific product's availability
`deleteProduct(uint256 _productID)`	removes the given product from the list of available products
`getProductData(uint256 _productID)`	returns a tuple containing the given product's data
`getAllProducts()`	returns the list of product's IDs

Tabella 24: Product contract methods

5.2.9 ProductTransaction

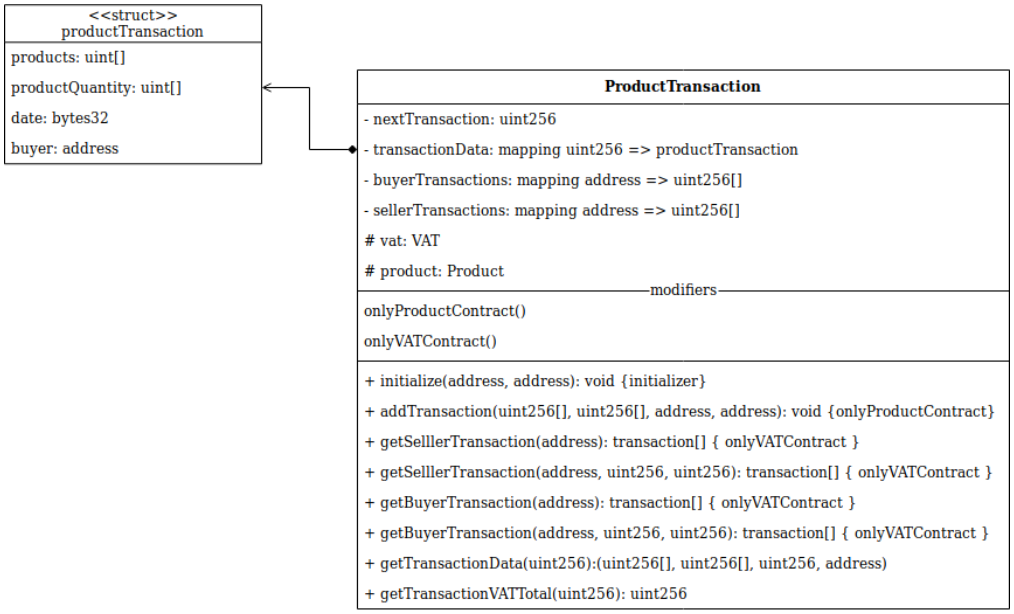

Figura 34: ProductTransaction contract

ProductTransaction is the contract in which the data of the transactions related to the buying and selling of products is stored, such as the bought products and their quantities, the date and the buyer.

5.2.9.1 Attributes

Attribute name	Description
`nextTransaction`	the ID of the next transaction to add
`transactionData`	a mapping that links the ID of a transaction to it's data
`buyerTransactions`	a mapping that stores all the transactions in which a certain user is the buyer
`sellerTransactions`	a mapping that stores all the transactions in which a certain user is the seller
`vat`	the instance of the deployed VAT's proxy
`product`	the instance of the deployed Product's proxy

Tabella 25: ProductTransaction contract attributes

5.2.9.2 Modifiers

Modifier signature	Description
`onlyProductContract()`	requires the calling contract to be Product
`onlyVATContract()`	requires the calling contract to be VAT

Tabella 26: ProductTransaction contract modifiers

5.2.9.3 Methods

Method signature	Description
`initialize(address _product, address _vat)`	initializes the contract
`addTransaction(uint256[] memory _productIDs, uint256[] memory _quantities, address _buyer, address _seller)`	adds a transaction storing the IDs of the bought producs, their quantities and the buyer and seller, which must be the same for all the products
`getSellerTransactions(address _seller)`	returns all the transactions that have the given business owner as a seller
`getSellerTransactions(address _seller, uint256 _fromDate, uint256 _toDate)`	returns all the transactions that have the given business owner as a seller in a given interval of time
`getBuyerTransactions(address _buyer)`	returns all the transactions that have the given business owner as the buyer
`getBuyerTransactions(address _buyer, uint256 _fromDate, uint256 _toDate)`	returns all the transactions that have the given business owner as the buyer in a given interval of time
`getTransactionData(uint256 _transactionID)`	returns the data of the transaction linked to the given ID
`getTransactionVATTotal(uint256 _transactionID)`	calculates the total VAT for the transaction linked to the given ID

Tabella 27: ProductTransaction contract methods

5.3 SoldinoAPI

To favor a greater division between front-end and back-end we decided to develop a JavaScript library that contains all the functions that allow the front-end to communicate with contracts written in Solidity called SoldinoAPI. Soldino API is in turn divided into modules, each containing methods concerning a different domain. These modules are:

5.3.0.1 Init

Function signature	Description
`getCubitContract()`	Retrieves the Cubit contract object
`getCurrentAccount()`	Retrieves the currentAccount selected from Metamask as an object
`getGovernmentContract()`	Retrieves the Government contract object
`getProductContract()`	Retrieves the Product contract object
`getProductTransactionContract()`	Retrieves the productTransaction contract object
`getUserManagerContract()`	Retrieves the UserManager contract object
`getVatContract()`	Retrieves the Vat contract object
`getVatContract()`	Retrieves the VatTransaction contract object
`getWeb3Instance()`	Retrieves the web3 Instance
`setupAccountChecking()`	Checks if the account has been changed from Metamask

Tabella 28: SoldinoAPI.Init methods

5.3.0.2 Users

Function signature	Description
`registerCitizen(name, surname, fiscalCode, mail)`	this method builds the JSON object containing the citizens's data to be sent to the function "registerUser()" in order to register a citizen
`registerBusinessOwner(businessName, location, VATNumber, CE)`	this method builds the JSON object containing the business owner's data to be sent to the function "registerUser()" in order to register a business owner
`getUserBalance()`	returns the caller's balance in Cubit
`getUserData()`	returns the caller's data by taking the user's hash and balance from the blockchain and get the JSON file from IPFS
`getGovernmentData()`	returns the caller's data by taking the balance from the blockchain
`failedLogin()`	takes care to return an error if an attempt of login went wrong
`login()`	allows a registered user to log in Soldino and returns the user's data
`getAllCitizens()`	returns the list of all citizens
`getAllBusinessOwners()`	returns the list of all confirmed business owners
`getAllRegisteredUsers()`	returns the list of all the registered users that contains all citizens and all the confirmed business owners
`confirmBusinessOwner(address)`	Lets the government confirm the given business
`getAllBusinessData()`	Gets a list of objects containing all businesses' data
`getAllCitizenData()`	Gets a list of objects containing all citizens' data

`getAllConfirmedBusinessOwnersData()`	Gets a list of confirmed businessOwner data
`getAllRegisteredUsers()`	Retrieves the list of all the registered users that contains all citizens and all the confirmed business owners
`getBusinessData(address)`	Retrieve BusinessOwner data from address
`getCitizenData(address)`	Retrieve citizen data from address
`isBusinessOwnerConfirmed(address)`	Checks if a the given business owner has been confirmed by the government
`registerUser(userData)`	Registers the given user by taking the user's data and parsing them into a JSON file. Once the JSON file has been created this will be stored in IPFS; this operation returns an hash that will be used to register the user inside the blockchain
`removeBusinessOwner(address)`	Removes the given business from the list of the business owners

Tabella 29: SoldinoAPI.Users methods

5.3.0.3 Cubit

Function signature	Description
`mint(amount)`	allows the government to mint the given amount of Cubit
`distributeToAll(amount)`	allows the government to distribute the given amount of Cubit to all registered users except those business owners that have not yet been confirmed by the government
`distributeToUsers(users, amount)`	allows the governmento to distribute the given amount of Cubit to the user contained in the given list

`setAllowance(spender, value)`	Allows to set the allowance to a specific user of the given value

Tabella 30: SoldinoAPI.Users methods

5.3.0.4 Helper

Function signature	Description
`getBytes32FromIpfsHash(ipfsListing)`	Return bytes32 hex string from base58 encoded IPFS hash, stripping leading 2 bytes from 34 byte IPFS hash
`getIpfsHashFromBytes32(bytes32Hex)`	Return base58 encoded IPFS hash from bytes32 hex string

Tabella 31: SoldinoAPI.Users methods

5.3.0.5 Products

Function signature	Description
`addProduct(productData)`	Adds a product to the caller's list of products
`buyProducts(products)`	Allows a user to buy a listi of products
`changeAvailability(productID, availability)`	Allows to modify the availability of a specified product
`deleteProduct(productID)`	Allows to delete a product from the caller's list of ones
`editProduct(productData)`	Allows to change the given product's details
`getAllOwnersProductsData()`	Returns the calle's list of products of products
`getAllProducts()`	Returns the list of all products
`getAllProductsData(ownerAddress)`	Returns the given address' list of product with relative data

`getProductData(productID)`	Returns the given product's data
`isValidProductID(productID)`	Checks if the given product is valid or not

Tabella 32: SoldinoAPI.Users methods

5.3.0.6 Orders

Function signature	Description
`getAllBuyerTransactions(address)`	Returns the list of transaction made by the address as a buyer
`getAllSellerTransactions(address)`	Returns the list of transaction made by the address as a seller
`getBuyerTransactions(address, fromDate, toDate)`	Returns the list of transaction made by the address as a buyer in a specific date range
`getSellerTransactions(address, fromDate, toDate)`	Returns the list of transaction made by the address as a seller in a specific date range
`getTransactionData(transactionID)`	Returns the given transaction's data

Tabella 33: SoldinoAPI.Users methods

5.3.0.7 Vat

Function signature	Description
`getBusinessOwnerVATHistory(address)`	Returns the caller's payment history
`getBusinessVATStatus(year, quarter, businessAddress)`	Returns the status of a given payment related to a specific quarter
`getQuarterVATBalance(year, quarter, businessOwner)`	Returns the VAT balance of a given quarter of a given business owner

`getTransactionData(transactionID)`	Returns the given transaction's data
`getTransactionVATTotal(transactionID)`	Returns the given transaction's VAT total
`getVATCompensations(fromYear, fromQuarter, toYear, toQuarter)`	Returns the caller's amount of VAT compensations of a specific quarter
`getVATDeductions(fromYear, fromQuarter, toYear, toQuarter)`	Returns the caller's amount of VAT deductions of a specific quarter
`getVATTransactionData(transactionID)`	Gets Vat compensation from a transaction
`payVATToBusinessOwner(year, quarter, businessOwner)`	Allows the government to pay VAT to a business owner
`payVATToGovernment(year, quarter)`	Allows a business owner to pay VAT to government

Tabella 34: SoldinoAPI.Users methods

5.4 IPFS

IPFS is a peer-to-peer distributed file system that connects all computing devices with the same system of files. Since storing data into the blockchain is highly expensive, we decided to use it to store all that data that is not necessary for any calculation of the Soldino business, but is just complementary information about the users.
The interaction between the blockchain and IPFS is handled by SoldinoAPI, which provides the set of instructions to store data and get the hash of the file stored.

6 License

A Glossary

A.1 Introduction

In this section it will be developed an internal glossary, in order to explain all technical terms used throughout this document in the illustration of Soldino's architecture.

A.2 Words

A

Action
JavaScript function dispatched every time the related event happens that is sent to the reducer in order to modify the *store*$_g$ of the application.

B

Blockchain
A blockchain is a growing list of records, called blocks, which are linked using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data.

C

Components
Components in *React*$_g$ are JavaScript functions or JavaScript classes that extend the *Component*$_g$ abstract class. Components are used for rendering the *UI*$_g$.

Component
Abstract class available in *React*$_g$ package used for creating new concrete components

E

ERC-20
ERC-20 is a technical standard used for smart contracts on the Ethereum blockchain for implementing tokens.
ERC-20 defines a common list of rules for Ethereum tokens to follow within the larger Ethereum ecosystem, allowing developers to accurately predict interaction between tokens. These rules include how the tokens are transferred between addresses and how data within each token is accessed.

Ethereum
Ethereum is an open-source, public, blockchain-based distributed computing platform and operating system featuring smart contracts (scripting) functionalities.

J

JSON
JavaScript Object Notation (JSON) is an open-standard file format that uses human-readable text

to transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value).

I

IPFS
IPFS is a peer-to-peer distributed file system that seeks to connect all computing devices with the same system of files.

M

Monolithic architecture
In software engineering, a monolithic application describes a single-tiered software application in which the user interface and data access code are combined into a single program from a single platform. A monolithic application is self-contained, and independent from other computing applications.

Microservice
Microservices are a software development technique, a variant of the service-oriented architecture (SOA) architectural style, that structures an application as a collection of loosely coupled services.

MVVM
Model–view–viewmodel (MVVM) is a software architectural pattern.
MVVM facilitates a separation of development of the graphical user interface from development of the business logic or back-end logic.

Model
The model part of the *MVVM*$_g$ pattern adopted for the *WebApp*$_g$ of Soldino.

Payload
Data carried by an *action*$_g$ in the form of parameters passed to the action itself.

R

React
JavaScript library for building user interfaces.

Redux
JavaScript library for managing the state of JavaScript applications.

Reducer
JavaScript function used in *Redux*$_g$ for checking the type of the *action*$_g$ dispatched and change the *store*$_g$ data accordingly to the *payload*$_g$ of the action.

Routes
All the paths available to reach Soldino's pages.

S

Solidity
Solidity is a contract-oriented programming language to write smart contracts. It is used for implementing smart contracts on various blockchain platforms. This is the language used for writing all smart contracts of Soldino.

Store
Redux object that holds all the data of the application.

T

Toolkit
In computer science a toolkit is a collection of tools used to make the development of complex software easier and uniform.

U

UI
UI (user interface) is a point of interaction between a computer and humans; it includes any number of ways of interaction (such as graphics, sound, position, movement, etc.) where data is transferred between the user and the computer system.

V

View
The view part of the $MVVM_{\text{g}}$ pattern adopted for the $WebApp_{\text{g}}$ of Soldino.

View
The ViewModel part of the $MVVM_{\text{g}}$ pattern adopted for the $WebApp_{\text{g}}$ of Soldino.

W

Web3
Web3 is a collection of libraries which allows the user to interact with a local or remote Ethereum node, using an HTTP, WebSocket or IPC connection.

WebApp
In computing, a web application or web app is a client–server computer program which the client (including the user interface and client-side logic) runs in a web browser.